Model-Based Testing with Graph Grammars

Vincent de Bruijn

August 13, 2012

Abstract

Graph Grammars have many structural advantages, which are potential benefits for the model-based testing process.This report describes the setup of a research project where the use of Graph Grammars in model-based testing is researched. The goal of the project is to create a system for automatic test generation from Graph Grammars and to validate this system. A graph transformation tool, GROOVE, and a model-based testing tool, ATM, are used as the backbone of the system. The system will be validated using the results of several case-studies.

Contents

1 Introduction

In software development projects, often limited time and resources are available for testing. However, testing is an important part of software development, because it decreases future maintainance costs [14]. Testing is a complex process and should be done often [19]. Therefore, the testing process should be as efficient as possible in order to save resources.

Test automation allows repeated testing during the development process. The advantage of this is that bugs are found early and can therefore be fixed early. A widely used practice is maintaining a *test suite*, which is a collection of tests. However, when the creation of a test suite is done manually, this still leaves room for human error [11]. Also, manual creation of test-cases is not time-efficient.

Creating an abstract representation or a *model* of the system is a way to tackle these problems. What is meant by a model in this report, is the description of the behavior of a system. Models such as state charts and sequence charts, which only describe the system architecture, are not considered here. A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. This leads to a larger test suite in a shorter amount of time than if done manually. These models are created from the specification documents provided by the end-user. These specification documents are 'notoriously error-prone' [13]. If the tester copies an error in the document or makes a wrong interpretation, the constructed model becomes incorrect.

The stakeholders evaluate the constructed model to verify its correctness. However, the visual or textual representation of large models may become troublesome to understand, which is referred to as the model having a low model transparency. The feedback process of the stakeholders is obstructed by low transparency models. Models that are often used are state machines, i.e. a collection of nodes representing the states of the system connected by transitions representing an action taken by the system. The problem in such models with a large amount of states is the decrease of model transparency. Errors in models with a low transparency are not easily detected.

A formalism that among other things can describe software systems is Graph Transformation. The system states are represented by graphs and the transitions between the states are accomplished by applying graph change rules to those graphs. These rules can be expressed as graphs themselves. A graph transformation model of a software system is therefore a collection of graphs, each a visual representation of one aspect of the system. This formalism may therefore provide a more intuitive approach to system modelling. Graph Transformation and its potential benefits have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]: "Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples." An informative paper on graph transformations is written by Heckel et al. [7]. A quote from this paper: "Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular."

1.1 Research setup

This report describes the setup of a project where the use of graph grammars in model-based testing is researched. In the introduction the motivation is given for using this type of modelling technique. The goal is to create a system for automatic test generation on graph grammars. If the assumptions that graph grammars provide a more intuitive modelling and testing process hold, this new testing approach will lead to a more efficient testing process and fewer incorrect

models. The system design, once implemented and validated, provides a valuable contribution to the testing paradigm.

Tools that perform statespace exploration on graph grammars and tools for automatic test generation already exist. Two of these tools will be used to perform these functions. The graph transformation tool GROOVE[1] will be used to model and explore the graph grammar. The testing tool developed by Axini[2] is used for the automatic test generation on *symbolic* models, which combine a state and data type oriented approach. This tool will be referred to in this report as Axini Test Manager (ATM).

1.2 Research questions

The research questions are split into a design and validation component:

1. **Design**: How can automatic test generation be done using graph transformation systems? In particular, how can ATM be used together with GROOVE to achieve this?

2. **Validation**: What are the strengths and weaknesses of using graph transformation systems in model-based testing?

The result of the design question will be one system which incorporates ATM and GROOVE. This system will be referred to as the GROOVE-Axini Testing System (GRATiS). In order to answer the first research question sufficiently, GRATiS should produce tests on the basis of a graph grammar and give correct verdicts whether the system contains errors or not.

The criteria used to assess the strengths and weaknesses are split into two parts: the objective and the subjective criteria. The objective criteria are the measurements that can be done on GRATiS, such as speed and statespace size. The subjective criteria are related to how easy graph transformation models are to use and maintain. The assessment of the latter criteria requires a human component. The criteria and methods for the assessment are elaborated in section 7.2.

GRATiS will be validated using case-studies. These case-studies are done with existing specifications from systems tested by Axini. Each case-study will have a graph grammar and a symbolic model which describe the same system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the model and the test process will then be compared as part of the validation. This includes the criteria mentioned in this section. This is explained in more detail in section 7.2.

1.3 Structure of the report

The structure of this document is as follows: the general model-based testing process is set out in section 2. Some basic concepts from first order logic are described in section 3, used as a framework for dealing with data in the symbolic models used in ATM. These models are then described in section 4. Section 5 describes graph grammars used in GROOVE. GROOVE and ATM are described in section 6. The method used to answer the research questions above is in section 7. Finally, the planning for the project is in appendix A.

2 Model-based Testing

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance to a model description of the system. The general setup for this process is depicted in

---

[1] http://sourceforge.net/projects/groove/
[2] http://www.axini.nl/

Figure 1. The specification of a system, given as a model, is given to a test derivation component which generates test cases. These test cases are passed to a component that executes the test cases on the SUT. Tests are executed by providing input/stimuli to the SUT and monitoring the output/response. The test execution component evaluates the test cases, the stimuli and the responses. It gives a 'pass' or 'fail' verdict whether the SUT conforms to the specification or not respectively.
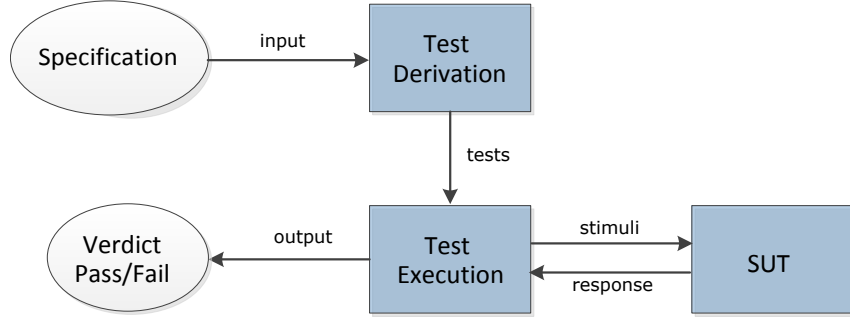

Figure 1: A general model-based testing setup

This type of model-based testing is called *batch testing* or *offline testing*. Another type of model-based testing is *on the fly* testing. The main difference is that no test cases are derived, instead a transition in the model is chosen and tested on the system directly. The general architecture for this process is shown in Figure 2. A tool for on-the-fly testing is TorX [24], which integrates automatic test generation, test execution, and test analysis. A version of this tool written in Java under continuous development is JTorX [2].
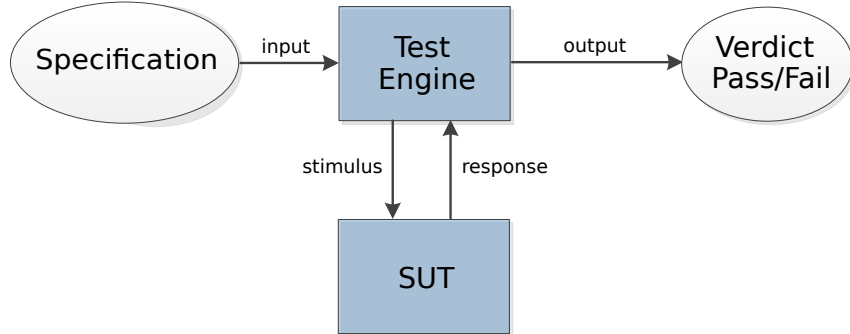

Figure 2: A general 'on-the-fly' model-based testing setup

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data. First, previous work on model-based testing is given. Then, two types of models are introduced. These are basic formalisms useful to understand the models in the rest of the paper. Finally, the notion of *coverage* is explained.

2.1 Previous work

Formal testing theory was introduced by De Nicola et al. [17]. The input-output behavior of processes is investigated by series of tests. Two processes are considered equivalent if they pass exactly the same set of tests. An important behavioural preorder of this research is the so called *must testing*, where the relevant tests are those to which the processes are inable to respond negatively to. This led to the possibility to observe deadlock by means of an unobservable transition,

called *refusal* testing [18]. The testing theory by De Nicola et al. was first used in algorithms for automatic test generation by Brinksma [3]. This led to the so-called *canonical tester* theory. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [25] and an algorithm for deriving a sound and exhaustive test suite from a specification in [26]. A good overview of model-based testing theory and past research is given in "Model-Based Testing of Reactive Systems" [12].

2.2 Labelled transition systems

A labelled transition system is a structure consisting of states with labelled transitions between them.

Definition 2.1. A labelled transition system is a 4-tuple $\langle Q, L, T, q0 \rangle$, where:

- Q is a finite, non-empty set of states

- L is a finite set of labels

- $T \in Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation

- $q0 \in Q$ is the initial state.

We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, \mu, q') \in T$. The informal idea of such a transition is that when the system is in state q it may perform action μ, and go to state q'.

2.3 Input-output transition systems

A useful type of transition system for model-based testing is the Input-Output Transition System (IOTS) by Tretmans [26]. Assuming that implementations communicate with their environment via inputs and outputs, this formalism is useful for describing system behavior. IOTSs have the same definition as LTSs with one addition: each label $l \in L$ has a type $t \in T$, where $T = \{input, output\}$. Each label can therefore specify whether the action represented by the label is a possible input or an expected output of the system under test.

An example of such an IOTS is shown in Figure 3a. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a question mark, the outputs are preceded by an exclamation mark. This system is a specification of a coffee machine. A test case can also be described by an IOTS with special pass and fail states. A test case for the coffee machine is given in Figure 3b. The test case shows that when an input of '50c' is done, an output of 'coffee' is expected from the tested system, as this results in a 'pass' verdict. When the system responds with 'tea', the test case results in a 'fail' verdict. The pass and fail verdicts are two special states in the test case, which are sink states, i.e., once in either of those the test case cannot leave that state.

Test cases should always reach a pass or fail state within finite time. This requirement ensures that the testing process halts.

2.4 Coverage

The number of tests that can be generated from a model is potentially infinite. Therefore, there must be a test selection strategy to maximize the quality of the tests while minimizing the time spent testing. Coverage statistics help with test selection. Such statistics indicate how much of the SUT is tested. When the SUT is a black-box, typical coverage metrics are state and transition coverage of the model [10, 16, 6].
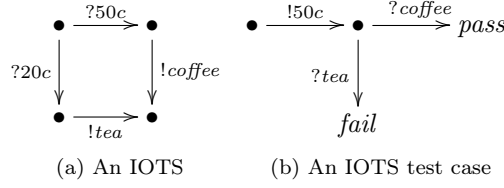
(a) An IOTS   (b) An IOTS test case

Figure 3: The specification of a coffee machine and a test case as an IOTS

As an example, let us calculate the coverage metrics of the IOTS test case example in 3b. The test case tests one path through the specification and passes through 3 out of 4 states and 2 out of 4 transitions. The state coverage is therefore 75% and the transition coverage is 50%.

Coverage statistics are calculated to indicate how adequately the testing has been performed [27]. These statisics are therefore useful metrics for communicating how much of a system is tested.

3 First Order Logic

Some basic concepts from first order logic are described here, used in the definitions of section 4. For a general introduction into logic we refer to [8].

Let $\mathcal{V}$ be a set of *variables*. *Terms* over V, denoted $\mathcal{T}(V)$, are built from a set of function symbols F and variables $V \subset \mathcal{V}$. Each $f \in F$ has a corresponding arity $n \in \mathbb{N}$. if $n = 0$ we call f a constant. We write $var(t)$ to denote the set of variables appearing in a term t. Terms $t \in \mathcal{T}(\emptyset)$ are called ground terms.

A *term-mapping* is a function $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{V})$. For sets V, W with $V \cup W \subset \mathcal{V}$, we write $\mathcal{T}(W)^X$ for the set of term-mappings that assign to each variable $v \in V$ a term $t \in \mathcal{T}(W)$, and to each variable $v \notin V$ the term v.

A *valuation* ν is a function $\nu : \mathcal{V} \mapsto \mathcal{U}$, where $\mathcal{U}$ is a non-empty set called a *universe*. An example of a universe is $\mathbb{N}$, the non-zero integers. Additionally, $\nu : tuple\langle \mathcal{V} \rangle \mapsto tuple\langle \mathcal{U} \rangle$, maps variables from $\mathcal{V}$ in a tuple to the respective values from $\mathcal{U}$ in a tuple.

An *evaluation* of $\mathcal{T}(\mathcal{V})$ is given by $\epsilon : (\mathcal{T}(\mathcal{V}), \nu : \mathcal{V}) \mapsto \mathcal{U}$. An evaluation $\epsilon : (\mathcal{T}(\mathcal{V}), \nu_{before} : \mathcal{V})$ of the terms in a term-mapping results in a set $X \subseteq \mathcal{U}$. A new valuation can then be constructed by: $\nu_{after} : \mathcal{V} \mapsto X$.

4 Symbolic Transition Systems

Symbolic Transition Systems (STSs) combine a state oriented and data type oriented approach. These systems are used in practice in ATM and will therefore be part of GRATiS. First, previous work on STSs is given. The definition of STSs and IOSTSs follow. An example of an IOSTS is then given. Next, the transformation of an STS to an LTS is explained and illustrated by an example. This transformation is useful when comparing STSs to systems that are not STSs. Finally, different coverage metrics on STSs are explained.

4.1 Previous work

STSs are introduced by Frantzen et al. [9]. This paper includes a detailed definition, on which the definition in section 4.2 is based. The authors also give a sound and complete test derivation

algorithm from specifications expressed as STSs. Deriving tests from a symbolic specification or *Symbolic test generation* is introduced by Rusu et al. [23]. Here, the authors use *Input-Output Symbolic Transition Systems* (IOSTSs). These systems are very similar to the STSs in [9]. However, the definition of IOSTSs we will use in this report is based on the STSs by [9]. A tool that generates tests based on symbolic specification is the STG tool, described in Clarke et al. [4].

4.2 Definition

An STS has *locations* and *switch relations*. If the STS represents a model of a software system, a location in the STS represents a state of the system, not including data values. A switch relation defines the transition from one location to another. The *location variables* are a representation of the data values in the system. A switch relation has a *gate*, which is a label representing the execution steps of the system. Gates have *interaction variables*, which represent some input or output data value. Switch relations also have *guards* and *update mappings*. A guard is a term t, where any evaluation on t with any valuation results in a value from $\mathbb{B} = \{true, false\}$. Such a term is denoted by $\mathcal{F}(\mathcal{V})$. The guard disallows using the switch relation when the evaluation of the term results in *false*. When the evaluation results in *true*, the switch relation of the guard is *enabled*. An update mapping is a term-mapping of location variables. After the system switches to a new location, the variables in the update mapping will have the value corresponding to the evaluation of the term they map to.

Definition 4.1. A Symbolic Transition System is a tuple $\langle L, l_0, \mathcal{L}, \imath, \mathcal{I}, \Lambda, \rightarrow \rangle$, where:

- L is a finite set of locations and $l_0 \in L$ is the initial location.

- $\mathcal{L}$ is a finite set of location variables.

- $\imath$ is a term-mapping $\mathcal{L} \rightarrow \mathcal{T}(\emptyset)$, representing the initialisation of the location variables.

- $\mathcal{I}$ is a set of interaction variables, disjoint from $\mathcal{L}$.

- Λ is a finite set of gates. The unobservable gate is denoted τ ($\tau \notin \Lambda$); we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\lambda \in \Lambda_\tau$, denoted $arity(\lambda)$, is a natural number. The parameters of a gate $\lambda \in \Lambda_\tau$, denoted $param(\lambda)$, are a tuple of length $arity(\lambda)$ of distinct interaction variables. We fix arity(τ) = 0, i.e. the unobservable gate has no interaction variables.

- $\rightarrow \subseteq L \times \Lambda_\tau \times \mathcal{F}(\mathcal{V} \cup \mathcal{I}) \times \mathcal{L} \mapsto \mathcal{T}(\mathcal{L} \cup \mathcal{I}) \times L$, is the switch relation. We write $l \xrightarrow{\lambda, \phi, \rho} l'$ instead of $(l, \lambda, \phi, \rho, l') \in \rightarrow$, where ϕ is referred to as the guard and ρ as the update mapping. We require $var(\phi) \cup var(\rho) \subseteq \mathcal{V} \cup param(\lambda)$, where var is the collection of the variables used in the given guard or update mapping.

An IOSTS can now easily be defined. The same difference between LTSs and IOTSs applies, namely each gate in an IOSTS has a type $t \in T$, where $T = \{input, output\}$. As with IOSTSs, each gate is preceded by a '?' or '!' to indicate whether it is an input or an output respectively.

4.3 Example

In Figure 4 the IOSTS of a simple board game is shown, where two players consecutively throw a die and move along four squares. The 'init' switch relation is a graphical representation of the variable initialization $\imath$. The defining tuple of the IOSTS is:

$\langle\{throw, move\}, throw, \{T, P, D\}, \{T \mapsto 0, P \mapsto [0, 2], D \mapsto 0\}, \{d, p, l\}, \{?throw, !move\},$
$\{throw \xrightarrow{?throw, 1<=d<=6, D \mapsto d} move, move \xrightarrow{!move, T=p \wedge l=(P[p]+D)\%4, P[p] \mapsto l, T \mapsto p\%2} throw\}\rangle$

The variables T, P and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The output gate !*move* has $param = \langle p, l \rangle$ symbolizing which player moves to which location. The input gate ?*throw* has $param = \langle d \rangle$

symbolizing which number is thrown by the die. The switch relation with gate ?*throw* has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate !*move* have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In Figure 4 the gates, guards and updates are separated by pipe symbols '|' respectively.
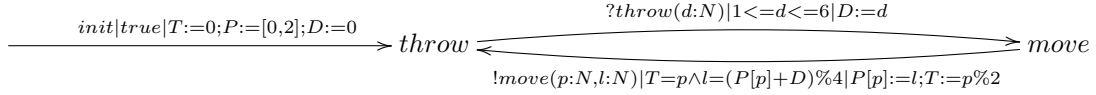
$$\xrightarrow{init|true|T:=0;P:=[0,2];D:=0} throw \underset{!move(p:N,l:N)|T=p\wedge l=(P[p]+D)\%4|P[p]:=l;T:=p\%2}{\overset{?throw(d:N)|1<=d<=6|D:=d}{\rightleftarrows}} move$$

Figure 4: The STS of a board game example

4.4 STS to LTS transformation

Consider an STS S and its transformation LTS L. The following defines a mapping between $\mathcal{L}$, $\mathcal{V}$, $\mathcal{I}$ and $\rightarrow$ of S to the states Q and transitions T of L.

- $\mu_l : (l \in L, \nu : \mathcal{V}) \mapsto q \in Q$
- $\mu_r : (r \in \rightarrow, \nu : \mathcal{I}) \mapsto t \in T$

Finding the topology of L is the next step of the transformation. Consider a switch relation r from location A to location B, a valuation of the location variables ν_l and interaction variables ν_i. Then, $\mu_l : (A, \nu_l)$ maps to a state q. If the result of the evaluation $\epsilon : (\phi{:}r, \nu_l \cup \nu_i)$ is *true*, then q is the source state of a transition t, such that $\mu_r : (r, \nu_i) \mapsto t$. ν_{l-new} is the new valuation of the location variables constructed by the evaluation of $\rho : r$. Then, the target state q' of t is the state mapped by $\mu_l : (B, \nu_{l-new})$. The label of t is a textual representation of λ of r and ν_i. Applying this rule for the topology to all locations, switch relations and concrete values for the variables, results in L. The start state $q0$ of L is the state mapped by $\mu_l : (l_0, \imath)$. All states not reachable from $q0$ are removed from L. When the number of possible valuations for $\mathcal{L}$ and $\mathcal{I}$ and the number of locations in an STS is considered to be finite, the transformation is always possible to an LTS with finite number of states.

An example of this transformation is shown in Figure 5. The label 'do(1)' in the LTS is a textual representation of the gate 'do' plus a valuation of the interaction variable 'd'. The transformation of a switch relation and concrete values to a transition is also called *instantiating* the switch relation. Another term we will use for a switch relation with a set of concrete data values is an *instantiated switch relation*.

4.5 Coverage

The simplest metrics to describe the coverage of an STS are the location and switch-relation coverage, which express the percentage of locations and switch relations tested in the test run. Measuring state and transition coverage of an STS is possible using the LTS resulting from the STS transformation. However, this metric is not always useful, because the number of states and transitions in the LTS depends on the number of unique combinations of concrete values of the variables in the STS. This is potentially very large. For example, when the guards of the switch relations in Figure 5a are removed, the transformation leads to an LTS with a state and transition for each possible value of an integer. It is often unfeasible to test every data value in the STS. The most interesting data values to test can be found by *boundary-value analysis* and *equivalence partitioning*. For an explanation of these terms we refer to [15]. Boundary-value analysis was found to be most effective by Reid [20] in fault detection.
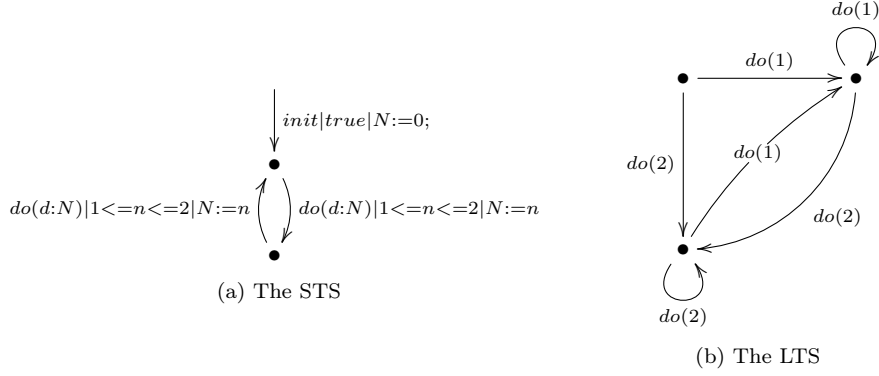
9

(a) The STS

(b) The LTS

Figure 5: An example of a transformation of an STS to an LTS

Data coverage expresses the percentage of data tested in the test run, considering data to be similar if located in the same partition and a better representative of the partition if located close to a boundary of the partition. There is no exact formula to calculate this percentage, however these considerations made by using equivalence partitioning and boundary-value analysis show which data should be tested to achieve a higher data coverage percentage.

5 Graph Transformation

A *graph grammar* is composed of a start graph and a set of transformation rules. The start graph describes the system in its initial state. The transformation rules describe what changes are made to the graph, resulting in a new graph which describes the system in its new state. The definition is as follows.

Definition 5.1. A graph grammar is a tuple $\langle G, R \rangle$, where:

- G is the start graph
- R is a set of graph transformation rules

The rest of this section is ordered as follows: first, graphs and graph morphisms are explained. This is then used to explain graph transformation rules, followed by the definition of a graph grammar. Then, the definition of a *Graph Transition System* (GTS) is given. An example of a graph grammar and a GTS is then given. Finally, a method for transforming a GTS to an STS is given. For a more detailed overview of graph grammars, we refer to [21, 7, 1].

5.1 Graphs & morphisms

Definition 5.2. A graph is a tuple $\langle L, N, E \rangle$, where:

- L is a set of labels
- N is a set of nodes, where each $n \in N$ has a label $l \in L$
- E is a set of edges, where each $e \in E$ has a label $l \in L$ and nodes *source, target* $\in N$

A graph H has an *occurrence* in a graph G, denoted by $H \to G$, if there is a mapping from the nodes and the edges of H to the nodes and the edges of G respectively. Such a mapping is called a *morphism*. An element e in graph H is then said to have an *image* in graph G and e is a *pre-image* of the image. A graph H has a *partial* morphism to a graph G if there are elements in H without an image in G.

5.2 Graph transformation rules

Definition 5.3. A transformation rule is a tuple $\langle LHS, NAC, RHS, M \rangle$, where:

- LHS is a graph representing the left-hand side of the rule
- NAC is a set of graphs representing the negative application conditions
- RHS is a graph representing the right-hand side of the rule
- M_{RHS} is a partial morphism of LHS to RHS
- M_{NAC} are partial morphisms of LHS to each $n \in NAC$

A rule R is applicable on a graph G if its LHS has an occurrence in G and $\nexists n \in NAC$ such that n has an occurence in G and $\forall e \in LHS$, if e has an image i in n and an image j in G, then j should be an image of j. After the rule application, all elements in LHS not part of M_{RHS}, i.e. they do not have an image in RHS, are removed from G and all elements in RHS not part of M_{RHS}, i.e. they do not have a pre-image in LHS, are added to G.

5.3 Graph transition systems

By repeatedly applying graph transformation rules to the start graph and all its consecutive graphs, a graph grammar can be explored to reveal a *Graph Transition System* (GTS). This transition system consists of *graph states* connected by *rule transitions*.

Definition 5.4. A graph transition system is an 8-tuple $\langle S, L, T, G, R, M_G, M_R, s0 \rangle$, where:

- S is a finite, non-empty set of graph states
- L is a finite set of labels. For each label $l \in L$, the arity of l, denoted $arity(l)$, is a natural number. The parameters of l, denoted $param(l)$, is a tuple of length $arity(l)$ of variables.
- $T \in S \times (L \cup \{\tau\}) \times S$, with $\tau \notin L$, is the rule transition relation. The parameters of a transition $t \in T$ with label $l \in L$, denoted $param(t)$, is a tuple of length $arity(l)$ of constants, such that $\Sigma : param(l) \mapsto param(t)$ is the valuation of the variables of the label.
- G is a set of graphs.
- R is a set of rules.
- M_G is a mapping $\forall s \in S.s \mapsto g \in G \wedge \nexists s' \in S.s \neq s' \wedge s' \mapsto g \in M_G$
- M_R is a mapping $\forall t \in T.t \mapsto r \in R \wedge \nexists t' \in T.t \neq t' \wedge t' \mapsto r \in M_R$
- $s0 \in s$ is the initial graph state.

We write $s \xrightarrow{\mu} s'$ if there is a rule transition labelled μ from state s to state s', i.e., $(s, \mu, s') \in T$.

These systems are very similar to LTSs. A GTS can be transformed to an LTS by omitting the graphs, rules, mappings and parameters on labels.

5.4 Example

Figure 6 shows an example of the start graph and one rule of a graph grammar. M_{RHS} maps the A and B nodes in LHS to the A and B nodes in RHS respectively. M_{NAC} maps the A node in LHS to the A node in both graphs in NAC. The a-edge in LHS is mapped to the a-edge in the first NAC. The LHS of the rule has an occurrence in the start graph, as the A and B node connected by the a-edge exists in both graphs. None of the graphs in the NAC have an occurrence in the start graph, because the C node does not exist in the start graph. The new graph after applying the rule is in Figure 6f.
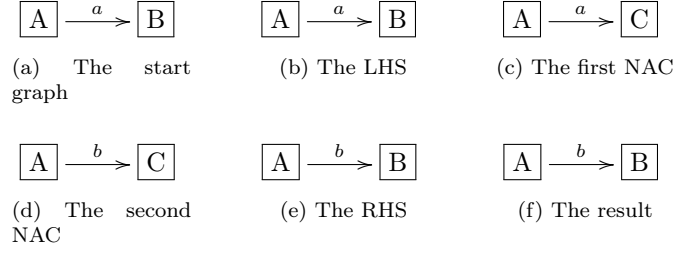
(a) The start graph   (b) The LHS   (c) The first NAC

(d) The second NAC   (e) The RHS   (f) The result

Figure 6: An example of a graph grammar

5.5 GTS to STS transformation

The method described here transforms a GTS to an STS. This STS is not *optimal*, i.e. it is reducible to an STS with fewer locations and switch relations. This method only serves as a proof used in section 7.1.2.

For each graph state $s \in S$ create a location $l \in L$. The start location $l_0 \in L$ is the location created by the state $s0 \in S$. Set $\mathcal{V} = \emptyset$ and $\imath = \emptyset$. For each label $l \in L$ create a gate $\lambda \in \Lambda$. For each $p \in param(l)$ create an interaction variable $i \in \mathcal{I}$ of the same type as p and add i to $param(\lambda)$. Set $arity(\lambda) = arity(l)$. For each rule transition $t \in T$ create a switch relation $r \in \rightarrow$, where:

- the source and the target locations of r are the locations created by the source and target graph states of the rule transition.

- the gate of r is the gate created by the label on the transition.

- for each $p \in param(l)$ and $i \in \mathcal{I}$ created by p create an $i = p$ expression. The ϕ of r can then be constructed by joining those expressions by the $\wedge$ operator.

- the ρ of r is empty.

An example of such a transformation is done using the LTS in Figure 5b. Consider this LTS to be a GTS where the states are graph states, the transitions are rule transitions with label 'do' with $arity(do) = 1$ and $param(do)$ the integers between brackets are The STS resulting from the transformation is in Figure 7.
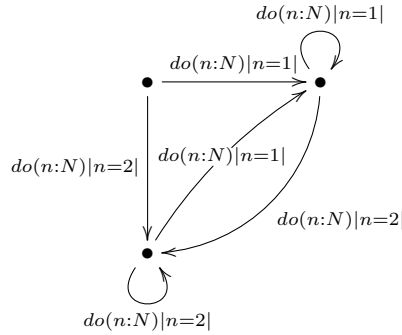

Figure 7: The basic STS resulting from the transformation of the LTS in Figure 5b

6 Tooling

6.1 ATM

ATM is a web-based model-based testing application, developed in the Ruby on Rails framework. It is used to test the software of several big companies in the Netherlands since 2006. It is under continuous development by Axini.

The architecture is shown graphically in Figure 8. It has a similar structure to the on-the-fly model-based testing tool architecture in Figure 2.
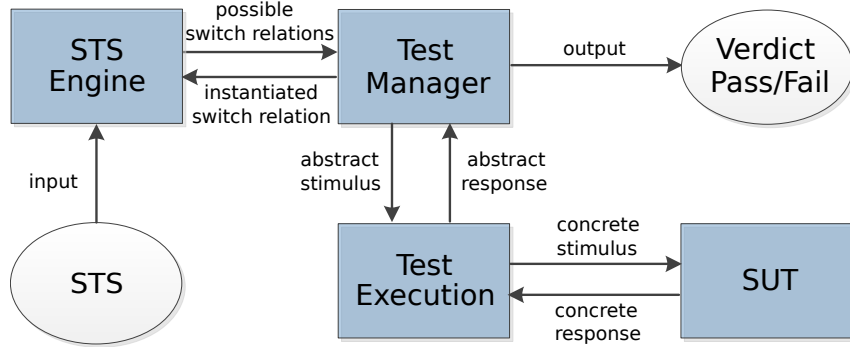

Figure 8: Architecture of ATM

The tool functions as follows:

1. An STS is given to an STS Engine, which keeps track of the current location and data values. It passes the possible switch relations from the current location to the Test Manager.

2. The Test Manager chooses an enabled switch relation based on a test strategy, which can be a random strategy or a strategy designed to obtain a high location/switch relation coverage. The valuation of the variables in the guard are also chosen by a test strategy, which can be a random strategy or a strategy using boundary-value analysis. The choice is represented by an instantiated switch relation and passed back to the STS Engine, which updates its current location and data values. The communication between these two components is done by method calls.

3. The gate of the instantiated switch relation is given to the Test Execution component as an *abstract stimulus*. The term abstract indicates that the instantiated switch relation is an abstract representation of some computation steps taken in the SUT. For instance, a transition with label 'connect?' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT. The communication between these two components is done by method calls.

4. The translation of an abstract stimulus to a concrete stimulus is done by the Test Execution component. This component provides the stimulus to the SUT. When the SUT responds, the Test Execution component translates this response to an abstract response. For instance, the Test Execution component receives an HTTP response that the TCP connect was succesful. This is a concrete response, which the Test Execution component translates to an abstract response, such as a transition with label 'ok!'. The Test Manager is notified with this abstract response. The communication between the Test Execution component and the SUT is done through sockets on both ends.

5. The Test Manager translates the abstract response to an instantiated switch relation and updates the STS Engine. If this is possible according to the model, the Test Manager gives a pass verdict for this test. Otherwise, the result is a fail verdict.

6.2 GROOVE

GROOVE is an open source, graph-based modelling tool in development at the University of Twente since 2004 [22]. It has been applied to several case studies, such as model transformations and security and leader election protocols [5].

The architecture of the GROOVE tool is shown graphically in Figure 9. A graph grammar is given as input to the Rule Applier component, which determines the possible rule transitions. An Exploration Strategy can be started or the user can explore the states manually using the GUI. These components request the possible rule transitions and respond with the chosen rule transition (based on the exploration strategy or the user input). The Exploration Strategy can do an exhaustive search, resulting in a GTS. The graph states and rule transitions in this GTS can then be inspected using the GUI.
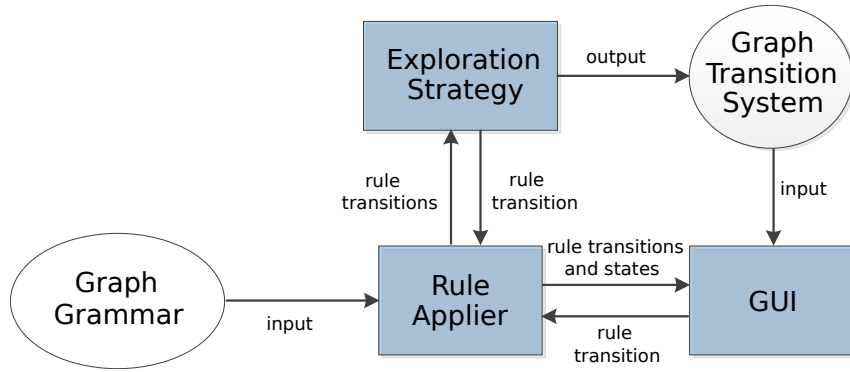

Figure 9: The GROOVE Tool

6.2.1 GROOVE example

The *LHS*, *RHS* and *NAC* of a rule in graph transformation tools, such as GROOVE, are often visualized together in one graph. The running example from Figure 4 is displayed as a graph grammar, as visualized in GROOVE, in Figure 10. Figure 10a is the start graph of the system.

The colors on the nodes and edges in the rules represent whether they belong to the *LHS*, *RHS* or *NAC* of the rule.

1. normal line (black): This node or edge is part of both the *LHS* and *RHS*.

2. dotted line (red): This node or edge is part of the *NAC* only.

3. thick line (green): This node or edge is part of the *RHS* only.

4. dashed line (blue): This node or edge is part of the *LHS* only.

The rules can be described as follows:

1. 10b: 'if a player has the turn and he has not thrown the die yet, he may do so.'

2. 10c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'

3. 10d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The *turn* flag on the **Player** node is a representation of a self-edge with label *turn*. The assignments on the **Die** node are representations of edges to integer nodes. The throws value assignment (:=) in the move rule is a shorthand for two edges: one edge in the *LHS* with label *throws* from

14

the **Player** node to an integer node with value i and another edge in the *RHS* with label *throws* from the **Player** node to an integer node with value $i - 1$. In the next turn rule, the *turn* edge exists in the *LHS* as a self-edge of the left **Player** node and in the *RHS* as a self-edge of the right **Player** node. In the same rule, the *throws* edge from the left **Player** node to an integer node only exists in the *LHS*. The number '0' in the top left of the **int** node in the throw rule indicates that this integer is the first parameter in $param(l)$, where l is the label on the rule transition created by applying the throws rule.
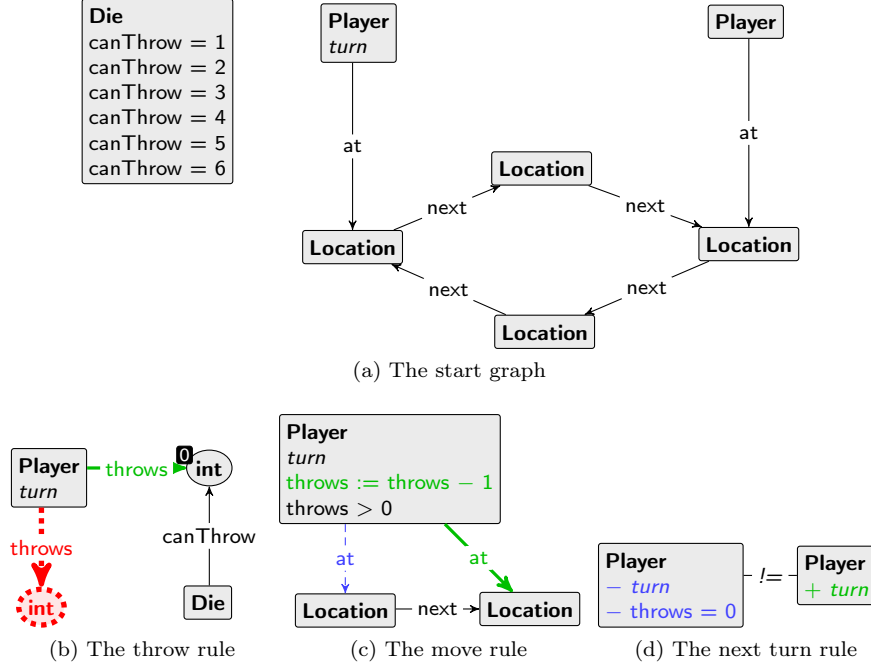
Figure 10: The graph grammar of the board game example in Figure 4

The graph is transformed after the rule is applied. The resulting graph after the transformation is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state. Figure 11 shows the GTS of one *throws* rule application on the start graph. The number on the label is the *parameter* of the label. State s_1 is a representation of the graph in Figure 10a. Figure 12 shows the graph represented by s_2.
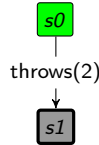
Figure 11: The GTS after one rule application on the board game example in Figure 10

6.3   Comparison of the examples

The models of the boardgame example in Figures 4 and 10 are very different. In this section the STS and the graph grammar of the example are compared.
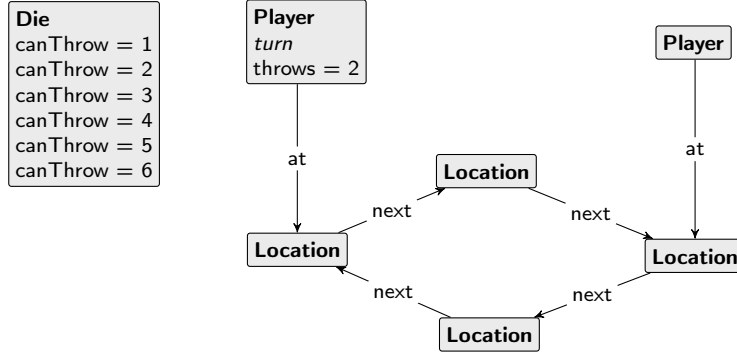
Figure 12: The graph of state $s2$ in Figure 11

6.3.1 Comparison of behavior

The GTS of the graph grammar of the boardgame example has a number of consecutive transitions when a player moves. The *move* rule puts the **Player** on the next **Location** and lowers the remaining 'moves' by one. This is different from the STS, which updates the location variable in one transition. The effect is that the behavior of both systems is different; one specifies the movement of a **Player** as: "Player p moves to Location l", the other as: "The Player with the turn moves to the next Location". Which behavior is required when this boardgame would be tested depends on the implementation of the game. However, to show the power of the GTS formalism, Figure 13 shows the graph grammar with the same behavior as the STS. It models the location as a variable and updates this variable in one transition. It also identifies the **Players** by giving them a number.

(a) The start graph
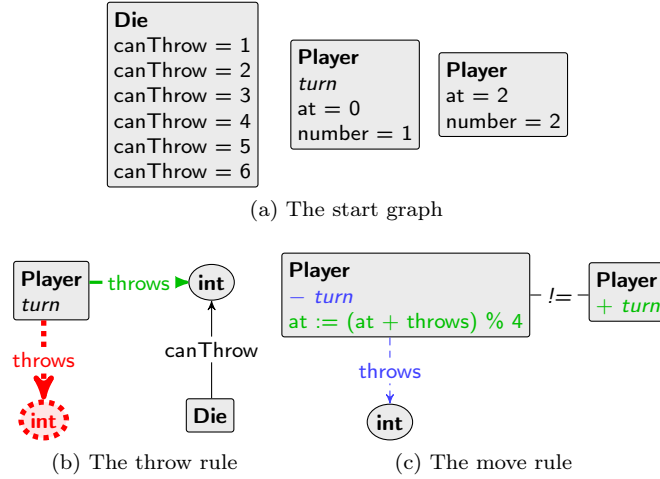

(b) The throw rule

(c) The move rule

Figure 13: Another graph grammar of the board game example

The new graph grammar loses many advantages by structuring it in this way: the overview of the board is gone, the rules are less visual and extending the locations in different directions is much harder. On the other hand, there are fewer rules and the graphs are more compact. However, when finding the GTS corresponding to this graph grammar, the labels of the transitions of that GTS do not reflect the 'move(p:N, l:N)' label of the STS. This should be done by marking the correct nodes as described in section 5.4. The problem is that the result of the equation in the 'move' rule is only derived when the rule is applied. Figure 14 shows a rule where the equation is shown graphically. The die roll, connected by the 'throws' edge, and the number of the **Location** the **Player** is at are added. The result is represented by the **int** node connected by the 'add' edge.

This result modulo 4 is represented by the **int** node connected by the 'mod' edge. This node is marked as the second parameter, the number of the **Player** is marked as the first parameter. This labels the transitions with which player moves to which location.
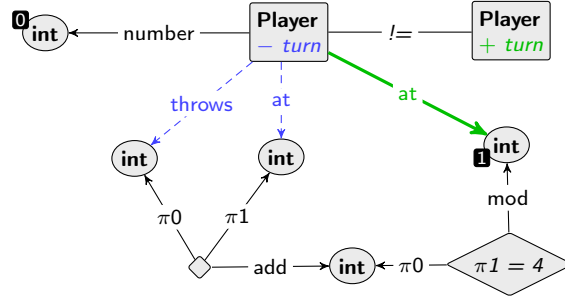

Figure 14: An alternative move rule

6.3.2 Comparison of transition systems

The GTS of a graph grammar can be computed using GROOVE and the STS can be transformed to an LTS. The two graph grammars and the STS of the board game example result in three transition systems which can be compared.

The graph grammar from Figure 10 generates a GTS with 32 states with 52 transitions, which can be seen visually in Figure 16. The graph grammar from Figure 13, using the 'move' rule from Figure 14, generates 224 states with 384 transitions, shown as GTS in Figure 17. The reason of the difference in number of states and transitions is that the board is circular: to the first graph grammar, the players being at locations 1 and 3 is the same as them being at locations 2 and 4. However, this is not the same to the second graph grammar. Also, for the first graph grammar it does not matter which **Player** node is at a **Location** node; they are the same apart from which **Player** has the 'turn'. As an example, consider the start graph in Figure 10. If both players throw a '1' and move to the next location, the state is as shown in Figure 15. Both states are symmetrical and therefore they are the same state. This leads to a *symmetry reduction* of the statespace for the first graph grammar.
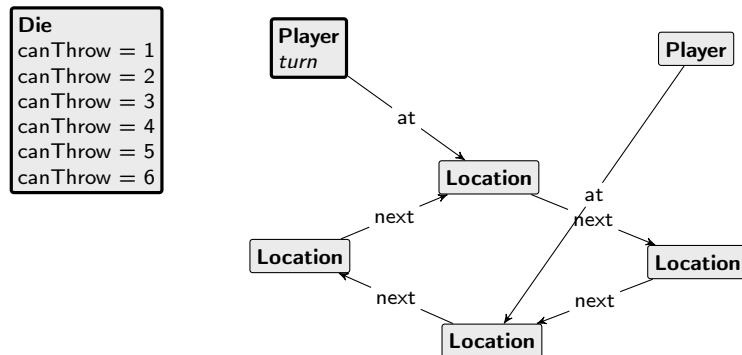

Figure 15: An example of a state symmetrical with the state in Figure 10

The LTS where the STS in Figure 4 is transformed to has 224 states and 384 transitions. This is calculated by taking all possibilities of the data values except for the die roll. This leads to 32 states ($4 \times 4 \times 2$). These 32 'throw' states each have 6 'throw?' transitions to a 'move' state, thus there are 192 'move' states. The 'move' states only have one transition back to a 'throw' state. There are $6 \times 32 + 192 \times 1 = 384$ transitions.
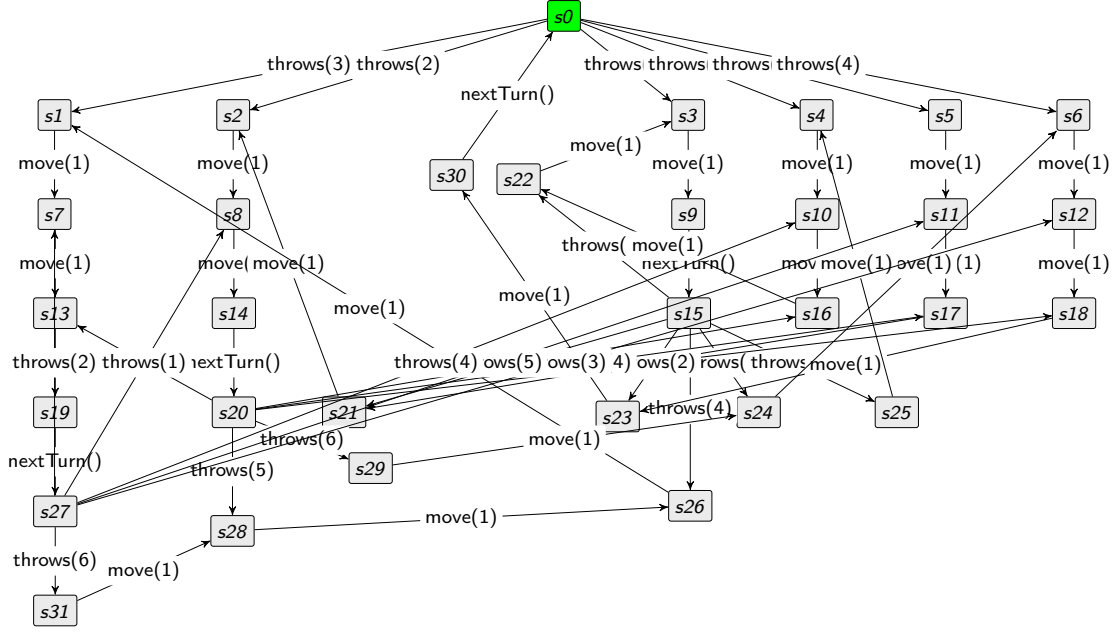
Figure 16: The GTS of the model in Figure 10

7 Research Method

7.1 GRATiS design

The specific method used to achieve the design goal is to use GROOVE as a replacement of the STS in ATM. Figure 18 shows this graphically. The **Test Manager** component communicates with an **STS Engine Interface**. This component obtains STS information from an **Exploration Strategy Interface** component, located at the GROOVE side. The **Rule Applier** component obtains the exploration strategy from the latter interface. The possible rule transitions are translated by the interfaces to possible switch relations for the Test Manager. The instantiated switch relation chosen is translated back to a rule transition by the interfaces. The exploration of the graph grammar is done *on the fly*; rule transitions are explored only when chosen by the Test Manager. This is a setup for an on the fly model-based testing tool with on the fly model exploration. Note that this setup allows other tools to also communicate with these interfaces.

7.1.1 Problems

The boardgame example revealed some problems, listed below, for the GRATiS design. These need to be tackled during the design phase of the project.

- Modelling data types such as integers and strings in GROOVE is problematic; the range of possible values has to be given explicitly. For example, it would be impossible to extend the die of the boardgame example such that it can throw any integer; each value of an integer would have to be defined separately.

- Coverage statistics cannot be calculated on GRATiS, if the model exploration is done on the fly. The number of states/locations and transitions/switch relations the model has when completely explored are not known, so a percentage cannot be derived.
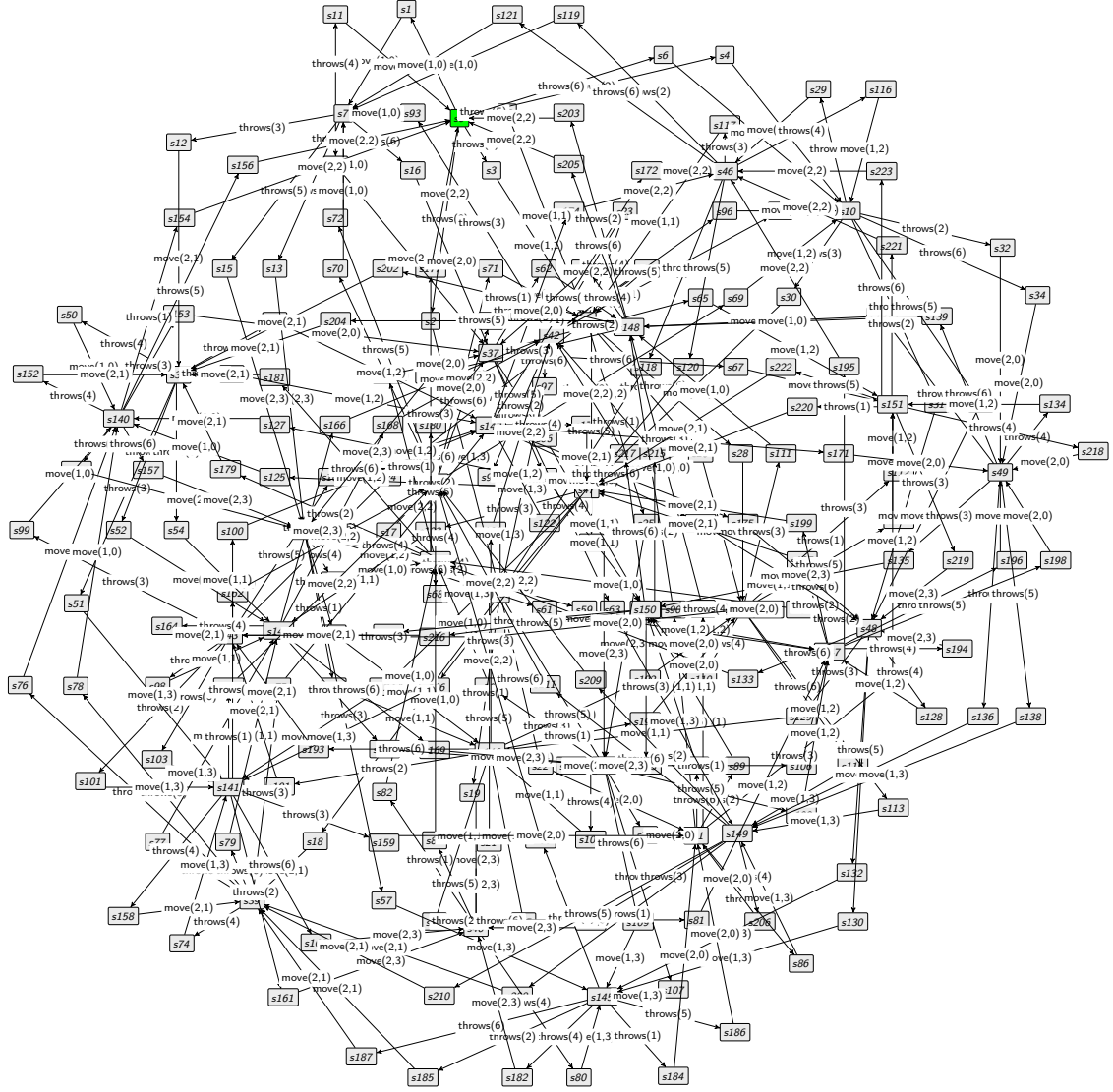
Figure 17: The GTS of the model in Figure 13

7.1.2 Initial design: GRATiS-Min

Before the problems in 7.1.1 are tackled and the design of GRATiS implemented, an initial design that minimally achieves the design goal is implemented. This initial design, named GRATiS-Min performs the following steps:

1. generate the GTS based on the graph grammar using GROOVE

2. transform the GTS to an STS with the method described in section 5.5

3. send the STS to ATM

4. perform the automatic test generation on the STS

This setup can be classified as on the model-based testing with *offline* model exploration.

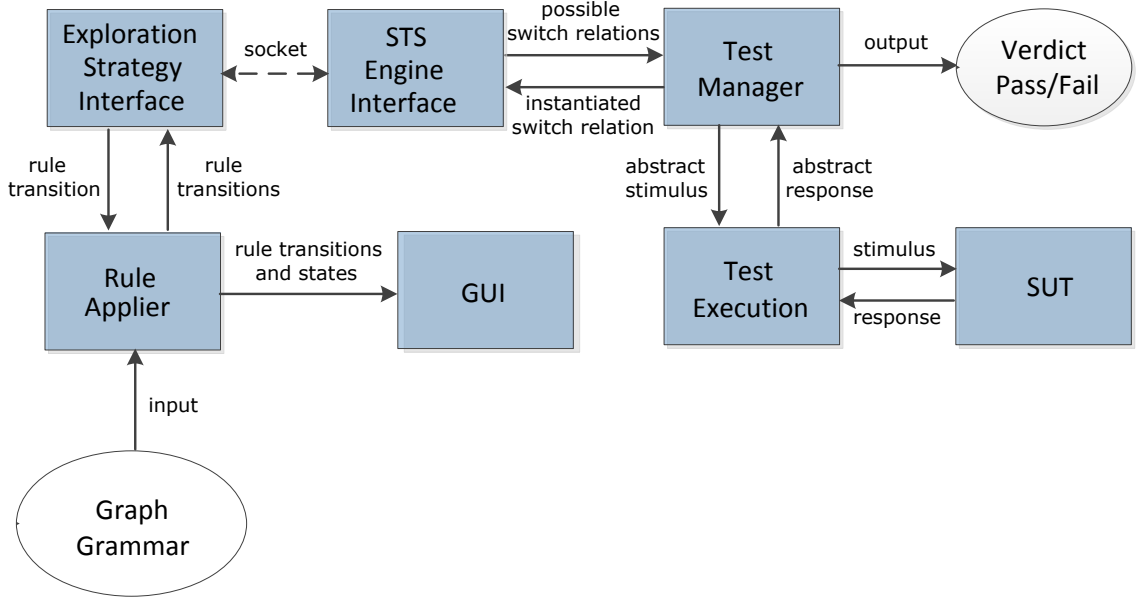There are a number of advantages to GRATiS-Min:

Figure 18: The GRATiS design: replacing the STS with GROOVE

1. It allows automatic test generation on models which GROOVE is able to fully explore. This has been done on some interesting systems, as described in section 6.2.

2. It is easy to implement. Only the transformation in section 5.5 and a basic protocol for transmitting an STS need to be implemented.

3. It can calculate coverage statistics, because the number of states and transitions are known. Note that the location and switch relation coverage is the same as graph state and rule transition coverage, as the transformation creates one location and one switch relation for each graph state and rule transition respectively. Therefore, we will call location and switch relation coverage statics not *relevant* for suboptimal STSs.

4. It already implements some features of GRATiS, such as the communication between GROOVE and ATM, which makes GRATiS easier to build.

There are also some disadvantages:

1. All states and transitions in the GTS need to be explored first. This is very costly in terms of time and memory.

2. The test strategies of ATM do not work on the suboptimal STS. First of all, the strategy designed to obtain a high location/switch relation coverage will attempt to achieve a high graph state/rule transition coverage. The coverage on the reduced, optimal STS may therefore have been higher, as the strategy is satisfied by choosing switch relations it has already chosen with different data values (note that these are different switch relations in the suboptimal STS). Secondly, boundary-value analysis cannot be used, as each switch relation has one specific set of data values needed to enable the switch relation.

7.1.3 Solutions

A tool with on-the-fly model exploration can not calculate coverage statistics, as clarified in section 7.1.1. However, it does not suffer the disadvantages of GRATiS-Min, depending on whether graph states can be transformed to locations with location variables and the rule transitions to

switch relations, such that the resulting STS is optimal. However, this transformation of a GTS to an optimal STS can also be applied to GRATiS-Min to solve the second disadvantage. This also makes location and switch relation coverage relevant. The first disadvantage can be solved by finding a transformation from a graph grammar to an STS, without constructing the GTS first. If such a transformation can be found, the resulting tool will be an on the fly model-based testing tool with offline model exploration and coverage statistics. If such a transformation cannot be found, this tool will be very costly in terms of time and memory. The alternative is an on the fly model-based testing tool with on the fly model exploration and no coverage statistics.

7.2 GRATiS Validation

There are two steps in the validation:

1. The boardgame example is tested. A SUT is made for this game and ATM is used to test this game. Then, GRATiS is used to test the boardgame. The results should reveal no errors, fail verdicts or other differences in the output of both tools. An intentional error is then made in the SUT and the process is repeated. Still no errors or differences are expected, but both tools should find the error and give a fail verdict.

2. Next, the assessment of the strengths and weaknesses of GRATiS is done by applying both tools to several case studies and comparing the results. The case studies are set out first and then the criteria for the comparison are given.

7.2.1 Case studies

Three case studies are planned. They are all real-life systems Axini has worked on:

- a self-scan register
- a navigation system
- a health-care system

The self-scan register is a machine that automates the purchase of products at a supermarket. A customer can put his products on a conveyer belt and the system automatically calculates the price of the products. Then the customer pays and gets a receipt. The navigation system is a GPS device with a route planner. It allows a user to enter a destination and the system plans the correct route accordingly from the location of the user. The health-care system is a medical device.

A GTS and an STS will be created for each system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the model and the test process will then be compared as part of the validation. The next two sections give the criteria for this comparison.

7.2.2 Objective criteria

As mentioned in section 1.2, the criteria for the comparison of ATM and GRATiS are split in two parts. The objective criteria that can be compared are:

1. The verdicts: same test cases should give same verdicts. When a verdict is different for the same test case in GRATiS and ATM, this might indicate an error.

2. The number of bugs found: one tool could generate smarter test-cases and find more bugs.

3. The coverage: generating smarter test-cases can also lead to a higher coverage.

4. The test cases generated per second: benchmarks will be done on how fast the tools generate test-cases.

5. The size of the statespace: benchmarks will be done on how much space the tools use.

7.2.3 Subjective criteria

Subjective criteria, such as the maintainability and extendibility of a model, are related to the usability of GTSs versus alternative formalisms, such as STSs. Evaluaiting such criteria requires an extensive experiment with a statistical significant set of human actors. It is out of the scope of this project to perform such an experiment in order to compare two modelling formalisms. In section 1 a motivation is given for the use of GTSs in model-based testing. The qualities of GTSs described there cover the usability of the GTSs.

References

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[2] Axel Belinfante. JTorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[3] E. Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing, and Verification VIII*, 1988.

[4] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46002-0_34.

[5] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 14(1):15–40, February 2012.

[6] Hasan and Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311 – 325, 1992.

[7] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1 SPEC. ISS.):187–198, 2006.

[8] M.R.A. Huth and M. Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[9] Tim A.C. Willemse Lars Frantzen, Jan Tretmans. Test generation based of symbolic specifcations. Technical report, Nijmegen Institute for Computing and Information Sciences (NIII), 2005.

[10] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[11] R. Busser M. Blackburn and A. Nauman. Why model-based test automation is different and what you should know to get started. *International Conference on Practical Software Quality and Testing*, 2004.

[12] Joost-Pieter Katoen Manfred Broy, Bengt Jonsson and Alexander Pretschner. *Model-Based Testing of Reactive Systems*. Springer-Verlag Berlin Heidelberg, 2005.

[13] Thomas J. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. *National Bureau of Standards, Special Publication*, 1982.

[14] Steve McConnell. Software quality at top speed. *Softw. Dev.*, 4:38–42, August 1996.

[15] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[16] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29:55–64, July 2004.

[17] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[18] Iain Phillips. Refusal testing. *Theoretical Computer Science*, 50(3):241 – 284, 1987.

[19] Martin Pol. *Testen volgens Tmap (in dutch, Testing according to Tmap)*. Uitgeverij Tutein Nolthenius, 1995.

[20] S.C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 64 –73, nov 1997.

[21] A. Rensink. Towards model checking graph grammars. In S. Gruner and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS), Southampton, UK*, volume DSSE-TR-2003-02 of *Technical Report*, pages 150–160, Southampton, 2003. University of Southampton.

[22] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[23] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40911-4_20.

[24] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[25] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[26] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.

[27] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

A   Planning

When	What	Deliverable
week 1-3 2 Jan - 22 Jan	Implementing basic interfaces for both tools. Creating SUT of boardgame example.	GRATiS-Min
week 4-6 23 Jan - 12 Feb	Setting up transformation rules for GTS-to-STS. Implementing transformation on GROOVE interface.	1) GTS-to-STS transformation rules. 2) improved GRATiS-Min.
week 7-10 13 Feb - 11 Mar	Implementing interfaces of GRATiS.	GRATiS without coverage statistics.
week 11-13 12 Mar - 1 Apr	Implementing location/switch relation coverage statistics for GRATiS.	GRATiS with location/switch relation coverage statistics.
week 14-16 2 Apr - 22 Apr	Creating graph grammars for each case study.	graph grammars of several software systems.
week 17 23 Apr - 29 Apr	Running the implemented system on the created models.	Measurements and comparison of both tools on objective criteria.
week 18-20 30 Apr - 20 May	Implementing data coverage statistics for GRATiS.	GRATiS with data coverage support.
week 21-23 21 May - 10 Jun	Finish writing thesis.	The final thesis.